EXHIBIT 10

What is a Good Nearest Neighbors Algorithm for Finding Similar Patches in Images?

Neeraj Kumar[1⋆], Li Zhang[2], and Shree Nayar[1]

[1] Columbia University
[2] University of Wisconsin-Madison

Abstract. Many computer vision algorithms require searching a set of images for similar patches, which is a very expensive operation. In this work, we compare and evaluate a number of nearest neighbors algorithms for speeding up this task. Since image patches follow very different distributions from the uniform and Gaussian distributions that are typically used to evaluate nearest neighbors methods, we determine the method with the best performance via extensive experimentation on real images. Furthermore, we take advantage of the inherent structure and properties of images to achieve highly efficient implementations of these algorithms. Our results indicate that vantage point trees, which are not well known in the vision community, generally offer the best performance.

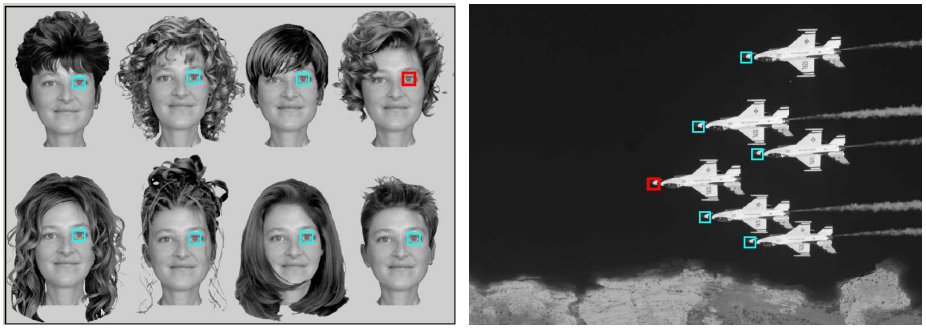
1 Introduction

Finding similar patches in images is a critical, yet expensive, step in a wide variety of computer vision tasks. For an image with N patches, an exhaustive search by scanning through the entire image for all N patches takes $O(N^2)$ time, which can take several minutes or even hours. For example, Figs. 1a and 1b each show results of a single search for patches (blue boxes) similar to the selected 21x21 patches (red boxes). Even though these searches were performed on relatively small images (roughly 800x600), they took 2.125 s and 1.375 s, respectively, to compute using brute-force scans. Finding the closest 10 neighbors for *all* patches in just these two images would take over 250 hours each!

However, by treating each image patch as a point in a high-dimensional space, we can use a Nearest Neighbors (NN) algorithm to compute the exact same results in a fraction of the time. Although a large number of general-purpose NN algorithms have been developed over the years [1–5], applying them directly to image search will not achieve the best results – images have distinct structure and unique properties that can be taken advantage of, to greatly improve search performance. In this work, we optimize many NN approaches specifically for finding similar patches in images and experimentally evaluate them to compare their performance. The following are the main contributions of our work:

– We describe several NN methods, some of which have not been widely used in computer vision. We restrict ourselves to exact approaches which guarantee that all NNs are found – approximate methods are not discussed.

⋆ Supported by the National Defense Science & Engineering Graduate Fellowship



(a) (b)

Fig. 1: The closest neighbors of the 21x21 patches outlined in red are shown in blue. The matching eyes in (a) took 2125 ms to compute using an exhaustive search, but only 1.25 ms (1700X faster) using our image-optimized implementation of the *vp*-tree, one of the methods discussed in this paper. Similarly, finding the nosetips in (b) took 1375 ms using brute force, but only 3.43 ms (401X faster) using the *vp*-tree.

- We use several techniques to significantly improve the efficiencies of the different NN algorithms (in some cases exploiting the inherent properties of images). These include precomputing or using look-up tables for computing distances, as well as using priority queues for certain types of searches.
- We evaluate the off-line (construction) and on-line (search) performances of the different methods using a set of real images. We study the behaviors of the algorithms with respect to various parameters, including patch size, image size, number of NNs, search range, and number of input images. We find that the *vantage point tree*, which has not been widely used in vision, has the best overall performance. (Our image-optimized implementation of this method computes identical results for the searches in Figs. 1a and 1b in only 1.25 ms and 3.43 ms – significantly faster than exhaustive search.)

The results of our work can be used to significantly reduce the computation times of several vision algorithms. For recognition, several approaches use local patches to match objects across images or videos [6–8]. In [9], similar patches within a neighborhood are used to find objects with similar shapes but very different appearances. Similarity search also lies at the core of many other vision problems. In [10], a weighted average of similar patches is used to denoise an image, while in [11], similar patches and a continuity constraint are used to synthesize a large image texture from a small exemplar. Patch similarity has also been used to perform super-resolution [12].

2 The Nearest Neighbors Problem

The Nearest Neighbors (NN) problem has been studied in many different fields, and is the topic of many surveys (see [13, 14] for recent examples). However,

these surveys focus mostly on taxonomizing a large variety of methods and do not offer performance results. In contrast, we focus on approaches most useful for finding similar image patches, and present many performance results.

The NN problem refers to finding the set of points $\mathbf{P_c}$ within a dataset $\mathbf{P}$ that are "closest" to a query point q, as measured by some distance function $d(p, q)$. Although points usually belong to some subset of $\mathbb{R}^n$ and d is often one of L_1, L_2, or L_∞, the only requirement typically imposed is that d is a true distance function, i.e., it satisfies

Symmetry: $d(a, b) = d(b, a)$;
Non-Negativity: $d(a, a) = 0$ and $d(a, b) > 0, a \neq b$;
Triangle Inequality: $d(a, b) \leq d(a, c) + d(b, c)$.

These properties form the basis of all the methods we will describe. In particular, the triangle inequality allows us to compute bounds on various distances without having to evaluate the distance function itself. To make maximum use of this property, all the approaches that we will look at organize the dataset into a hierarchical tree data structure. In these trees, the data points are stored in the leaf nodes and the internal nodes serve only to prune the list of leaves returned by a query. Particular methods differ in what is stored at each internal node and how the tree is constructed, but there are several commonalities between them.

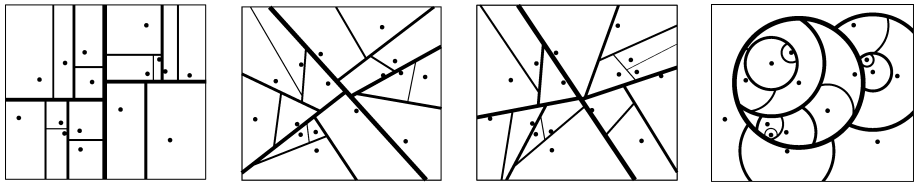
2.1 Tree Construction

Tree construction begins by assigning all points to the root node, and then recursively partitioning the points into one of several children of the node (the number of children, how they are chosen, and how the points are partitioned all vary depending on the particular method). This process continues until some termination criteria are met. Two common criteria are the maximum leaf size (the leaf contains fewer than a given number of points) and the maximum leaf radius (all points within the leaf are contained within a hyperball of given radius). Either criterion, or some combination of both, can be used for each tree type.

2.2 Tree Searches

There are two types of searches that we would like to perform:
ϵ-close neighbors (ϵ-NN): Find the set of points $\mathbf{P_c} \subset \mathbf{P}$ such that $p_i \in \mathbf{P_c} \Leftrightarrow d(p_i, q) \leq \epsilon$. This is also called *range search* in the existing literature.
k-nearest neighbors (k-NN): Find the set of k points $\mathbf{P_c} \subset \mathbf{P}$ such that $\forall p_i \in \mathbf{P_c}$ and $\forall p_j \notin \mathbf{P_c}, d(p_i, q) \leq d(p_j, q)$.

We now describe how NN searches are typically run on a given tree (see [13] for more detailed explanations). To perform an ϵ-NN search, the tree is traversed from the root, recursively exploring all children that intersect a hyperball of radius ϵ around the query point. Determining the list of intersecting children is done using the triangle inequality and some data stored in each node (dependent on the particular method). Once a list of leaf nodes are found, each contained



(a) kd-Tree (b) PCA Tree (c) Ball Tree (d) vp-Tree

Fig. 2: The partitioning of a 2D point set using different types of nearest neighbor trees, all with a maximum leaf size of 1 and a branching factor of 2. Line thickness denotes partition order (thicker lines were partitioned first). Note the very different structures created by the methods, which result in very different search speeds.

data point must be verified as being within ϵ of the query. This is required because many leaves might be too close to be pruned by the triangle inequality, but still contain many points that are farther away than ϵ. This step is usually the most expensive one because it requires a large number of distance computations.

To perform k-NN searches, we could simply run a number of ϵ-NN searches with increasing ϵ, but this is a wasteful strategy. Instead, a better approach is to first locate the leaf node where the query point would be contained, and then use backtracking to explore nearby nodes. We maintain a sorted list of the k closest points found so far, as well as the maximum radius around the query point that has been fully explored. Skipping nodes that are farther away than the distance of the kth closest point found so far, we add each candidate node's points to our list, sorted by distance. This process has the effect of reducing the distance of the kth neighbor. When this distance becomes less than the maximum explored radius, the search is complete (because all remaining points must be at a greater distance than the current maximum).

Note that while ϵ-NN search performance does not depend on the order in which nodes are explored (because all matches must be found), search order greatly affects the performance of k-NN searches. Since the distance of the kth element of the list is used to prune candidate nodes, the faster we reduce this distance, the more nodes we prune. We will discuss how to take advantage of this observation in Sect. 4.3.

3 Methods for Finding Nearest Neighbors

We now describe various methods for finding nearest neighbors. To give a more intuitive idea of how the different methods operate, a simple 2D dataset partitioned using each of the methods is shown in Fig. 2.

3.1 *k*d-Trees

Despite having been invented over three decades ago, Bentley's kd-tree [1] remains one of the most commonly used algorithms for finding nearest neighbors

today. Tree construction is simple: At each node, the points are recursively partitioned into two sets by splitting along one dimension of the data, until one of the termination criteria is met. The important choices to be made in tree construction are determining which dimension to split on and the split value. Choosing the dimension with maximum variance leads to smaller trees [15], while the split value is usually chosen to be the median value along the split dimension. This results in a balanced partitioning of points into axis-aligned hyper-rectangles, which get smaller in regions with many points (e.g., Fig. 2a).

The kd-tree exhibits many favorable properties and has proven to be quite efficient in practice for low-dimensional data. However, it has a few drawbacks:

1. The number of neighbors for each leaf node grows exponentially with dimension, causing search to quickly devolve into a linear scan.
2. The node divisions are always axis-aligned, regardless of the data distribution. This often results in poor search performance.

In real applications, the first problem is typically skirted by relaxing the requirement that all close neighbors be found. The Best Bin First (BBF) approach [16] is one such technique. It is based on the observation that the vast majority of the neighboring cells usually do not contain a nearest neighbor. It therefore searches the candidate cells in ascending order of their distance to the query, and terminates the search early to save computations. In [6], it is claimed that this method produces 95% of the correct neighbors at 1% the cost of an exhaustive search for one particular application. However, we note that this does not *guarantee* that all nearest neighbors are found, which can cause problems for various applications (e.g., as described in [17]).

3.2 PCA Trees

Sproull [2] attempts to remedy the axis-alignment limitation of kd-trees by applying Principal Components Analysis [18] at each node to obtain the eigenvector corresponding to the maximum variance and splits the points along that direction. This is equivalent to rotating the points about their mean such that their maximum variance now lies along the primary axis. We have implemented this variant as well, calling it the *PCA Tree*. For an example of what a tree partitioned using this method looks like, see Fig. 2b.

3.3 Ball Trees

The kd-tree and its variants can be termed "projective trees," meaning that they categorize points based on their projection into some lower-dimensional space. In contrast, all our remaining methods are "metric trees" – structures that organize points based on some metric defined on pairs of points. Thus, they don't require points to be finite-dimensional or even in a vector space.

The first type of metric tree that we will look at are *ball trees* [3]. In their original form, each node's points are assigned to the closest center of the node's

two children. The children are chosen to have maximum distance between them, typically using the following construction at each level of the tree. First, the centroid of the points is located, and the point with the greatest distance from this centroid is chosen as the center of the first child. Then, the second child's center is chosen to be the point farthest from the first one.

The resulting division of points can be understood as finding the hyperplane that bisects the line connecting the two centers, and perpendicular to it (e.g., see Fig. 2c). Note that in this construction, there is no constraint on the number of points assigned to either node and the resulting trees can be highly unbalanced. While unbalanced trees are larger (and take longer to construct) than their balanced counterparts, this does not mean that they will be slower to search. On the contrary, such trees might be significantly faster if they capture the true distribution of points in their native space.

3.4 *k*-Means

While the previous description of ball trees is probably familiar to members of the machine learning community, vision researchers will no doubt have noticed its similarity to the k-means method [19]. This algorithm also assigns points to the closest of k centers, although it does so by iteratively alternating between selecting centers and assigning points to the centers until neither the centers nor the point partitions change. The resulting structure is equivalent to a Voronoi partition of the points [20], which simplifies to the hyperplane described in the previous section for the case of $k = 2$.

As originally described, the k-means method is a simple non-hierarchical clustering method that requires careful selection of both k and the initial centers to avoid local minima and bad partitions. Linde et al. extend this method to a hierarchical structure [4] where k now defines the branching factor between successive levels of the tree. This variant, which is the one we have implemented, has faster construction and search times because fewer distance evaluations need to be performed at each level. We also note that if the centers are initialized using the procedure described in the previous section, very few iterations have to be run for the centers to converge.

3.5 Vantage Point Trees

We turn now to a metric tree that uses a single "ball" at each level – the vantage point tree (*vp*-tree) [5]. Rather than partitioning points on the basis of relative distance from multiple centers (as was the case with ball trees and k-means), the *vp*-tree splits points using the absolute distance from a single center. This approach can be visualized as partitioning points into "hypershells" of increasing radius, e.g., Fig. 2d. The center of each node can be chosen randomly, as the centroid of the points, or as a point on the periphery (to maximize the distance between points). The number and thickness of the "hypershells" can also be chosen in various ways, and we explore them experimentally in Sect. 5. As we shall see, these methods perform the best for finding similar image patches.

3.6 Other Methods

Another common NN method [21] is based on using the L_∞ norm to quickly find all points within the hypercube of size 2ϵ around the query point. While this method works very well for applications which require strong correspondences between individual dimensions of points, it is less suited for cases where the overall distance (e.g., as measured using the L_2 metric) is more important. This is because in higher dimensions the volume of a hyperball of radius ϵ is much smaller than the volume of the enclosing hypercube. Thus, only a small fraction of the returned points will actually be within the search distance required.

Finally, we mention a class of methods that attempt to solve a relaxed version of the NN problem. Rather than finding the closest points, they allow the algorithm to return points that are within a factor of $(1+\epsilon)$ of the true closest distance, for some $\epsilon \geq 0$. This "approximate nearest neighbor" problem has proven slightly easier to attack, and there exist many approaches to this problem (e.g., the ANN [22] and Locality Sensitive Hashing [23] libraries). A recent work [24] provides a performance analysis of these methods, albeit applied to SIFT [6] descriptors, not image patches. However, they do not cover all of our optimizations, nor do they consider the *vp*-tree, which we found to be most effective. Finally, exact methods are preferred for many applications, where the potential of missing real results or getting too many false results can cause problems.

4 Efficient Implementation of Algorithms

Since our primary goal in this work is to use NN methods for finding similar patches in images, we can exploit various properties of images to develop very efficient implementations of these algorithms.

4.1 Computing L_p Norms using Lookup Tables

The definition of what constitutes points as being "close" is critical. One commonly-used family of metrics are the L_p norms. For points $\mathbf{a}$ and $\mathbf{b}$ of dimensionality D, these are defined as $d(\mathbf{a},\mathbf{b}) = \left(\sum_i^D |a_i - b_i|^p\right)^{\frac{1}{p}}$. The L_p norms for $p=1$ (Manhattan distance) and $p=2$ (Euclidean distance) are especially common when comparing image patches. Here, we take advantage of the fact that images typically contain 8-bit quantized values. We use a 2-dimensional lookup table to store the result of $|a-b|^p$ for all possible values of a and b. This can be done during program initialization and saves D subtractions and $D\cdot(p-1)$ multiplications per distance function evaluation (converting them to array lookups, which are much faster on most hardware). Likewise, the expensive p-th root calculation can be stored in a lookup table if D and p are small enough.

4.2 Pre-Calculating Distances Within Each Leaf

During searches, a large portion of time is spent in evaluating the distance function for each point in returned leaf nodes. Once again, we take advantage of

the triangle inequality. For each leaf, we precompute distances of each contained point to the leaf center, and store the points in increasing order of distance. At search time, we use the triangle inequality to find the upper and lower bounds of the list of points that are within our chosen distance and only calculate the distance for these points. While this might appear to be equivalent to adding one more level to a tree during construction, observe that this optimization is run at search time and is thus more flexible – it can be applied for different search distances ϵ, regardless of the maximum leaf radius chosen during construction. This simple procedure helps immensely. On average, we reduce the number of distance calculations in ϵ-NN searches by 38.5%.

4.3 Priority Queues for *k*-NN Searches

For k-NN searches, the order in which nodes are explored is extremely important. Therefore, we maintain a priority queue of nodes to visit, sorted by minimum distance from the center of each node to the query point. We first traverse the tree down to the leaf node corresponding to the query point, adding all possible branches at each node to our queue. Then, we repeatedly pick the closest node from our queue and expand it, again adding its children to our queue. This process continues until the closest node from the queue is farther away than the distance of the kth neighbor found so far. The benefit of this optimization is that we explore the closest nodes to our query first, which in turn fills our queue faster with a tighter distance bound. Although this is similar in spirit to the BBF criteria [16] mentioned earlier, note that we still guarantee exact results. Also, our optimization can be applied to all tree types, not just k-d trees.

5 Performance Evaluation

All of the methods we have described have similar asympototic search times. However, in practice, their actual performance is dominated by the structure of the constructed trees. This structure is determined by the distribution of the input data. It is well known that images exhibit complex distributions which are very different from simple uniform or Gaussian ones [25]. Since there are no analytical ways to compute the search performance of trees built from image patches, we empirically measure performance on a set of 15 real-world images.

5.1 Methodology

Thumbnails of the images from our dataset (mostly collected from flickr.com) are shown in Fig. 3. They were chosen to represent a wide variety of image types, e.g., indoor, outdoor, portraits, etc. Note that even with this small number of test images, there are still millions of patches to search through. To reduce computation times, we downsampled all images to smaller sizes (longest side to 640 pixels) and converted them to grayscale. We also removed all uninformative "flat" patches from the images, using the procedure described in Appendix A.

Fig. 3: The image dataset used in the performance evaluation.

We use the number of distance comparisons as our cost unit for all tests, since this is typically the most expensive basic operation. To put cost values into perspective, we show the ratios of distance function evaluations performed in a naive brute force scan to those performed during search on a given tree type. When comparing search speeds, note that the brute force cost is independent of the query and the values of k or ϵ used for search. So, it is the relative search speeds of different methods that matters more than the exact numbers.

5.2 Comparison of Tree Types

The first task is to establish which tree type has the best construction and search performance. The two main parameters guiding tree construction are the maximum leaf size and the branching factor (or bin size), and so we construct many trees by sampling this 2D parameter space. At each parameter setting, we build a separate tree for each image and each tree type. Since our main goal at this point is just to compare the different NN methods, we use only non-flat 7x7 patches of each image. (We perform in-depth tests with different patch sizes in the next section.) For each tree, we then run ϵ- and k-NN searches for each patch of the same image ($\epsilon = 38.5$ and $k = 20$). Searching for every patch ensures that we do not introduce systematic biases into our results due to sampling issues. Also, the performance results are averaged over all trees built using the same parameters to get more accurate results.

Table 1 summarizes the construction and search performance results for all tree types. Note that we tried splitting vantage point trees using either constant branching factors or constant bin sizes, marked as *vp*-tree (k) and *vp*-tree (δ), because the two have different performance characteristics (other tree types did not show significant differences). This table allows us to pick the best parameter settings for each tree type.

To more easily compare the search performance of different tree types, we show the construction and search results for each tree in Figs. 4a and 4b, respectively, using their best parameter settings in Table 1. Note that *k*d-trees require no distance computations during construction, and so their construction cost shows up as 0. Also, PCA trees require many iterations of eigenvector decomposition, making their actual construction cost much higher than shown. We see that *k*d-trees can be built the fastest, followed by *vp*-trees. However, the search results in Figure 4b show that *k*d-trees are the slowest to search, while

Table 1: Comprehensive results. The splits are given as branching factors for ball trees, k-means, kd-trees, and vp-trees (k), and as bin sizes for PCA trees and vp-trees (δ). Construction costs are given as average number of distance calculations per patch. Search speeds are given as improvement over brute-force search. Optimal values are shown in **bold**.

Method	Leaf Size	Split	Cons. Cost	ε-NN Search Speed	k-NN Search Speed
Ball Tree	2	2	76.52	14.23	0.50
		3	65.81	14.74	0.62
		4	65.20	14.19	0.68
	16	2	69.80	**46.23**	**15.84**
		3	56.61	45.37	15.41
		4	56.26	44.17	15.42
	128	2	53.30	44.77	15.37
		3	46.44	44.31	14.87
		4	**46.33**	44.64	15.14
k-Means	2	2	303.92	4.47	0.50
		4	289.93	6.17	0.69
	16	2	271.60	19.20	7.62
		4	270.40	20.98	8.36
	128	2	**221.37**	35.09	12.69
		4	228.73	38.90	13.89
		8	298.72	**42.70**	**15.20**
PCA Tree	2	2	6.59	22.10	**10.26**
		4	8.25	25.50	9.81
		8	7.79	17.70	4.42
	16	2	5.50	26.60	10.13
		4	7.55	**27.72**	9.63
		8	7.61	17.41	4.24
	128	2	**4.32**	25.11	8.00
		4	6.24	25.80	8.37
		8	6.99	17.69	4.34

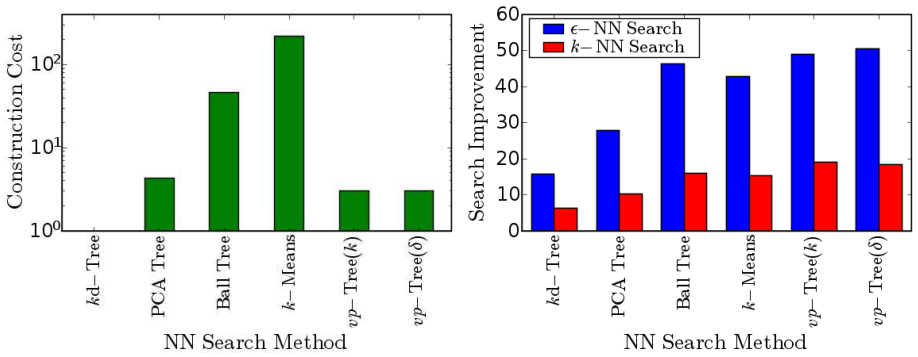
Method	Leaf Size	Split	Cons. Cost	ε-NN Search Speed	k-NN Search Speed
kd-Tree	2	1	**0.00**	4.45	1.88
	16	1	**0.00**	14.69	6.23
	128	1	**0.00**	**15.82**	**6.31**
vp-Tree (k)	2	2	16.28	38.54	1.69
		4	9.07	38.48	1.89
		8	7.49	34.83	1.57
		16	6.28	31.55	1.50
	16	2	13.97	44.66	15.84
		4	7.13	46.07	16.49
		8	5.00	**48.95**	17.61
		16	4.02	41.95	**19.12**
	128	2	11.05	33.80	11.85
		4	5.95	36.75	12.72
		8	4.00	37.50	12.53
		16	**3.00**	38.13	12.28
vp-Tree (δ)	2	2	5.71	45.29	10.27
		4	9.30	42.09	16.36
		8	9.60	36.96	16.94
	16	2	4.36	**50.45**	**18.27**
		4	7.94	49.36	18.21
		8	8.78	39.85	15.72
	128	2	**3.01**	44.25	13.05
		4	5.85	42.55	13.32
		8	6.64	36.23	12.31

vp-trees and ball trees are the fastest. Although the slow search performance of kd-trees in high dimensions is well known [21], we would like to draw the reader's attention to the low construction cost and fast search performance of our image-optimized version of the vp-tree, a method which is not in widespread use in the vision community. We believe its superior performance for both construction and search is due to the following two properties:

- Only one distance function evaluation is required at each level of the vp-tree compared to multiple ones for ball trees and k-means.
- The "hypershell" partitions of vp-trees can capture clusters of similar image patches using fewer nodes than the "hyperplane" partitions of other methods. This results in smaller trees with leaf nodes at a lower average height.

5.3 Performance Along Various Dimensions

We now want to see how search performance varies as different parameters are changed, holding the rest constant. Since it would be infeasible to run these tests on all tree types at all parameter settings, we choose the best-performing trees from our comparison in the previous section. For k-NN tests, we use the vp-tree



(a) Summary of Construction Costs (b) Summary of Search Performance

Fig. 4: Summaries of the (a) construction costs and (b) search performance for different types of NN trees. The construction costs are given as average number of distance computations per input patch, plotted on a logarithmic scale. The search speeds are given as ratio of time taken for brute force scan versus time taken using the given tree type. All numbers are for best parameter settings (see Table 1 for full listings). Note that although ball trees and *vp*-trees have similarly good search performance, the latter has a much smaller construction cost.

(k) with both the max leaf node size and branching factor set to 16. For ϵ-NN tests, we use the *vp*-tree (δ) with max leaf node size of 16 and bin size of 2.

Figure 5a shows search performance as a function of search distances ϵ and k, on the top and bottom, respectively. Note that both are plotted on a log-log scale, and so the linear portions of the graphs indicate an exponential drop-off in performance as either k or ϵ are increased. This results from the fact that the triangle inequality becomes progressively less useful as we search for increasingly dissimilar patches. It suggests that the search range should be kept as small as possible to obtain the best performance. Applications which require large search ranges will hence have to pay a steep penalty in terms of search speed.

Figure 5b plots the search performance as a function of patch size, on a semilog scale. To ensure a fair comparison of search distances at different patch sizes WxW, the actual search range ϵ used was computed as $\epsilon = W \cdot \epsilon'$, where the "average per-pixel distance" ϵ' was empirically chosen to be 5.5. We can see that as our patches get bigger, ϵ-NN search performance improves, but k-NN search performance detoriates. These are both effects of the increasing distances between patches in higher dimensions. Thus, a fixed "average per-pixel distance" will contain fewer points when the patch size increases, causing ϵ-NN search speeds to improve. However, the greater distance between patches also means that the k most similar patches will be at greater distances from the query – making the triangle inequality less useful for pruning patches.

So far, we have used single images to construct and search each of our trees. This is useful if we need to search for patches within only one image. However,

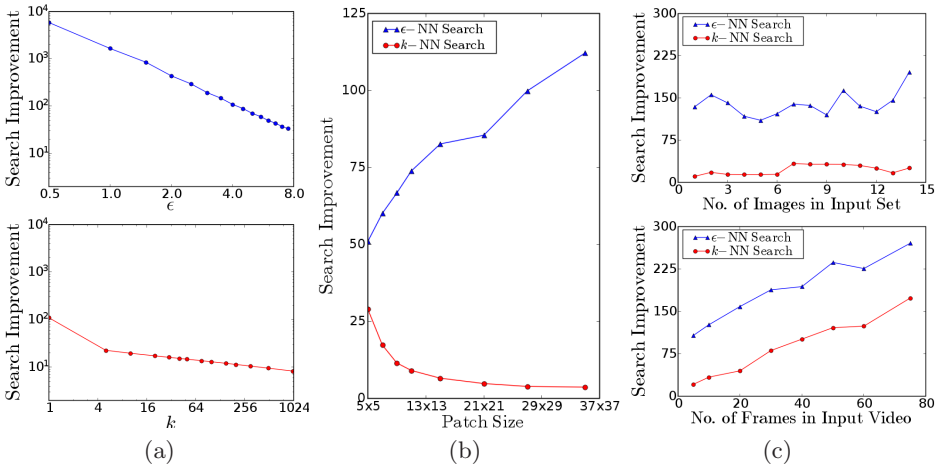


Fig. 5: Search performance of the *vp*-tree plotted as a function of (a) search distances in log-log, for (top) ϵ-NN and (bottom) k-NN searches; (b) patch size in semi-log; and (c) number of input images (top) and video frames (bottom). The exponential performance drop-off in (a) suggest that for high performance, small search ranges should be used (application allowing). As patch sizes go up, (b) shows k-NN searches slowing down and ϵ-NN searches speeding up – higher dimensionality causes the closest k neighbors to become harder to distinguish from other patches, but there are also less patches within the same "average per-pixel distance." In (c), we see that since the images in the test set did not have enough similarity for the *vp*-tree to take advantage of, the performance does not increase with the number of images (top graph). In contrast, performance improves dramatically with the number of video frames since frames taken from the same video clip tend to have strong similarity (bottom graph).

for many applications we would like to find patches across multiple images or video frames. Figures 5c top and bottom show the search performance for each of these cases, respectively. These results were computed by using a subset of the input images to build a *vp*-tree, and searching using all patches from one of the remaining images. In the case of multiple images, we arbitrarily chose the subsets from our set of images (as shown in Figure 3). We constructed several different subsets, and averaged their performance results. The search performance for this test varies drastically as we increase the number of images used (especially for ϵ-NN searches). This suggests that there was not much similarity between the images in our test set. Thus, the *vp*-tree's search performance scales roughly as that of brute force searches. Note, however, that *vp*-tree searches are still significantly faster than brute force.

The video datasets were constructed by sampling a 30fps, 5 second video at fixed intervals to use as inputs for building a *vp*-tree. As in the previous case, this tree was then searched using one of the other frames from the video. The steady increase in the video search speeds confirms our hypothesis about

Table 2: Summary of results. The *vp*-tree performs well in all respects.

Method	Construction Performance	ε-NN Search Performance	*k*-NN Search Performance
*k*d-Tree	Excellent	Poor	Poor
PCA Tree	Poor	Fair	Fair
Ball Tree	Fair	Excellent	Excellent
k-Means	Poor	Good	Good
vp-Tree	Excellent	Excellent	Excellent

similarity leading to better search performance. These frames share much more similarity than a set of arbitrary images, and the vantage point tree is able to take advantage of this fact. Thus, finding similar patches from a set of images becomes progressively faster as the dataset size grows, especially if the images have some similarity. In the future, it would be useful to see if large sets of arbitrary images contain enough similarity to show similar increases in performance.

6 Concluding Remarks

In this work, we have extensively evaluated a number of different approaches for solving the nearest neighbors problem. In particular, we have focused on using these approaches for finding similar image patches. Since this is a common step in a wide range of applications ranging from object recognition and texture synthesis to image denoising and compression, solving this problem efficiently will lead to faster solutions for all of these disparate tasks. Our main findings about using various NN methods for finding similar patches can be summarized as follows (also see Table 2):

- Vantage point trees have the best overall construction and search performance. Furthermore, their increasing speeds with larger datasets makes them very powerful for applications that involve large images or videos.
- Ball trees have similar search speeds, but much longer construction times.

References

1. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Comm. ACM **18**(9) (1975) 509–517
2. Sproull, R.F.: Refinements to nearest-neighbor searching in k-dimensional trees. Algorithmica **6**(4) (1991) 579–589
3. Omohundro, S.M.: Five balltree construction algorithms. Technical Report 89-063 (1989)
4. Linde, Y., Buzo, A., Gray, R.M.: An algorithm for vector quantizer design. IEEE Transactions on Communications **28**(1) (January 1980) 84–94
5. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. Symposium on Discrete algorithms (1993) 311–321

6. Lowe, D.: Distinctive image features from scale-invariant keypoints. IJCV (2003)
7. Sivic, J., Zisserman, A.: Video google: A text retrieval approach to object matching in videos. In: ICCV, Washington, DC, IEEE Computer Society (2003) 1470
8. Nister, D., Stewenius, H.: Scalable recognition with a vocabulary tree. CVPR (2006)
9. Shechtman, E., Irani, M.: Matching local self-similarities across images and videos. CVPR (2007)
10. Buades, A., Coll, B., Morel, J.M.: A non-local algorithm for image denoising. CVPR (2005)
11. Efros, A.A., Freeman, W.T.: Image quilting for texture synthesis and transfer. SIGGRAPH (2001)
12. Freeman, W.T., Jones, T.R., Pasztor, E.C.: Example-based super-resolution. IEEE Comput. Graph. Appl. **22**(2) (2002) 56–65
13. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys **33**(3) (2001) 273–321
14. Shakhnarovich, G., Darrell, T., Indyk, P.: Nearest-Neighbor Methods in Learning and Vision. MIT Press (2006)
15. Bentley, J.L.: K-d trees for semidynamic point sets. Symposium on Computational geometry (1990) 187–197
16. Beis, J.S., Lowe, D.G.: Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. CVPR (1997)
17. Hays, J., Efros, A.A.: Scene completion using millions of photographs. ACM Transactions on Graphics (SIGGRAPH) **26**(3) (2007)
18. Hotelling, H.: Analysis of a complex of statistical variables with principal components. Journal of Educational Psychology **24** (1933) 417–441
19. MacQueen, J.: Some methods for classification and analysis of multivariate observations. Symposium on Mathematical Statistics and Probability (1967) 281–297
20. Edelsbrunner, H.: Algorithms in combinatorial geometry. Springer-Verlag, New York, NY (1987)
21. Nene, S.A., Nayar, S.K.: A Simple Algorithm for Nearest Neighbour Search in High Dimensions. IEEE Transactions on Pattern Analysis and Machine Intelligence **19**(9) (Sep 1997) 989–1003
22. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. J. ACM **45**(6) (1998) 891–923
23. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. Symposium on Theory of Computing (1998) 604–613
24. Mikolajczyk, K., Matas, J.: Improving descriptors for fast tree matching by optimal linear projection. ICCV (2007)
25. Ruderman, D.L.: The statistics of natural images. Network: Computation in Neural Systems **5** (1996) 517–548
26. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. CVPR (2001)

A Low-Variance Patch Removal

We remove "flat" patches, which are those with low variance: $\sigma^2 < \sigma^2_{min}$, and all pixels close to the mean: $L_\infty(patch, \mu) < dist_{min}$. Here, μ and σ^2 are the mean and variance of the patch of size WxW. We empirically define $\sigma^2_{min} = 3.7 \cdot W$ and $dist_{min} = 5$. This test can be efficiently evaluated using integral images [26].